Practical Ajax Projects with Java™ Technology

Frank W. Zammetti

Apress®

Practical Ajax Projects with Java™ Technology

ISBN-13 (pbk): 987-1-59059-695-1

ISBN-10 (pbk): 1-59059-695-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Chris Mills
Technical Reviewer: Herman van Rosmalen
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editor: Liz Welch
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositor: Lynn L'Heureux
Proofreader: Linda Seifert
Indexer: Brenda Miller
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 8

The Organizer: Get Yourself Organized Now!

In this chapter, we'll build what is commonly called a PIM application, or Personal Information Manager. This application, which we'll call The Organizer, will include the ability to write notes to yourself, set up tasks to accomplish, store all your contacts, and make appointments. We'll learn about about another real web application framework, WebWork. We'll use a very popular Ajax library named Prototype for our Ajax functionality.

Requirements and Goals

The requirements for The Organizer are actually rather straightforward, since PIM applications are a fairly common thing. So, without further ado, our goals are as follows:

- First, we'll allow for multiple user accounts, so we'll need all the requisite account management functions, including creating, updating, and deleting accounts. Each user will maintain his or her own accounts, and we'll not have any administrative-type functionality on top of that (an obvious enhancement suggestion!).
- When a user logs in, they will start at a Day At A Glance screen, which will list those tasks and appointments for the current day.
- The Organizer will consist of four basic units of functionality: notes, tasks, contacts, and appointments.
- For notes, we should be able to record a subject and some text. That is really about it!
- For each task, we should be able to record a subject, a due date, a priority rating (high, normal, or low), a status (complete or incomplete), and some comments about the task.
- For contacts, we'll have a fair amount of data we can capture, including home phone, address, fax number, cell number, spouse, children, and so forth. We'll have much the same data for work information, with some extras like assistant, manager, title, company, and so forth. Lastly, we'll have essentially duplicates of most of these as *other*, so that for instance you can have a home address, a work address, and some other address, and likewise for most pieces of information.

- For appointments, we of course need to be able to create them, and record the subject of the appointment, what date it is, what time it starts and ends, and some comments about it.
- For appointments, we'll provide four different views: day view, week view, month view, and year view. We should be able to select a date to use as the basis for any of these views.
- We should be able to not only create notes, tasks, contacts, and appointments, but also modify and delete existing instances.
- We want to use a real framework for this application, not something we invent ourselves.

As you can see, these goals and requirements are not anything special as far as PIM applications go. But we have enough there to make life interesting, so let's figure out exactly how we are going to do it.

How We Will Pull It Off

One of the things I would like to touch upon in this project that I have not previously talked about is the methodology behind building this application. I want to describe how I went about putting it together, especially because it was done in an extremely short amount of time.

I approached this project with a service-oriented architecture (SOA) mentality. Basically, I identified all the individual functions that would be needed, and proceeded to code each as a separate, independent service. Especially when you are working with an Ajax application, this is a fairly natural way to look at things because you can visualize all the code that runs on the client as being the application itself, while the functions you call on the server are services that the application needs to do its work.

So, instead of thinking of pages, and a navigation flow through them, I thought about what functions are required for notes, for instance. Well, clearly we need to be able to list notes, so that is one service. We need to be able to create a note, so that is another service. It would be nice to be able to update an existing note, so there is another service. Finally, being able to delete a note makes sense, so that is yet another service.

Once I had done that bit of planning, I created a page to test each function. These were plain HTML pages with nothing fancy in them. For instance, the test page to create a note was

```
<html>
  <head>
    <title>The Organizer</title>
 </head>
  <body>
    Create Note
    <br><br>
    <form action="noteSave.action">
      subject: <input type="text" name="subject"><br>
      <textarea cols="40" rows="15" name="text"></textarea>
      <input type="submit" value="Create Note">
    </form>
  </body>
<html>
```

So, I wound up with 30 or so of these types of test pages, with little to no navigation between them (there had to be some for things like logon functionality, but by and large they were completely separate). I knew that the final application would not use these; in fact, it would probably not use anything remotely similar. At that point, I had not yet decided whether the server would return XML, HTML, or something else.

And that is precisely the important point! This application differs from all the others in this book in that every single response from the server to an Ajax request is rendered via JSP. We have seen this before, but this is the first application where that is the sole mechanism by which responses are created. The great thing about this is that you can decide how to return the response later; it becomes merely an implementation detail that can be deferred. This, coupled with the SOA approach, means that your application can change forms very easily, and you do not have to decide everything up front, which for me is the way I prefer to work (for me, early coding *is* design many times… I can quickly determine what will work and what will not that way).

Once I got to the point where all the individual pages worked with the server-side services, I then created the main JSP that would be the client-side code of the application. For each of the test JSPs, I ultimately decided to use them to return straight HTML, so I then coded them for real (which took only a few minutes per JSP, since the basic outline was already there). Once that was done, it was just a matter of the plumbing on the client side to glue all these discrete services and their responses together and turn it into a coherent whole.

I very much recommend this approach. Do not even think about Ajax initially; simply determine the discrete functions your application needs, the services the server must provide, and go to it. Once all the services are implemented, *then* you can begin to think about what the application looks like and how those services get cobbled together to form the larger whole. If you have JSPs (or some other flexible rendering technology like Velocity, for instance), then you do not even have to initially think about what the server returns for each service. I have found this approach works very well.

With that out of the way, we'll look at the four main tools we'll use to make this happen: Prototype, WebWork, HSQLDB, and Spring JDBC.

Prototype

Prototype is one of the most popular JavaScript libraries out there. Notice I did not call it an Ajax library; it is much more than just Ajax. It is the basis of many Ajax libraries and is integrated into many frameworks, so it is definitely worth taking a look at.

There is an often-stated criticism of Prototype—that it is dangerous because it extends basic JavaScript objects. This can have unintended side effects in other JavaScript code. There are examples on the Web of iterations of arrays not working because the `Array` object is extended, among other problems. Although this may be true, the simple fact is that Prototype is used by many other libraries with no bad side effects, so the case may be overstated. Then again, it may not! The point is that it is something you should be aware of. Some of the basic JavaScript objects, including `Object`, `Number`, `Function`, `String`, `Array`, and `Event`, are extended by Prototype and are therefore different than they usually are. While the intent is that these changes should not affect existing code, nor should it cause any problems for new code, there does seem to be instances where it can and does cause problems.

Prototype introduces a number of shortcuts to JavaScript. For instance:

```
$("newNote").style.display = "none";
```

What exactly is going on here? Simply put, `$()` is a function that is used in place of the ubiquitous `document.getElementById()`. There are a number of variations on this theme, including `$F()`, which gets a named form element, `$A()`, which converts its single argument to an array, `$H()`, which converts objects into enumerable `Hash` objects that resemble associative arrays, and `$R()`, which is simply shorthand for writing new `ObjectRange(lowerBound, upperBound, excludeBounds)` (the `ObjectRange` is an object that represents a range of values, with upper and lower bounds). You will see the `$()` function used often in this project; the others are not used.

The other important part of Prototype, for this project, is the `Ajax` object. This is where all the Ajax functionality lives. We'll use two functions of this object in this project: `Request()` and `Updater()`. The difference is that `Request()` just sends the request you set up, and then calls a function you specify to handle the response. `Updater()` is designed for perhaps the most common Ajax use case: updating an element on the page (a `<div>`, more often than not) with the server's response. Here is an example of the `Request()` function:

```
new Ajax.Request("myServer.action", {
  method : "post",
  postBody : Form.serialize($(myForm)),
  onSuccess : function(resp) {
   alert("Server returned to us");
  }
});
```

So, we call this function, passing it a number of arguments. The first argument is simply the URL to submit to, some resource named `myServer.action` in this case (assume we have this mapped to a servlet, for example). We specify we want a POST done, and then we populate the `postBody`. To do this, we use a very neat function of Prototype: `Form.serialize()`. This function accepts a reference to a form, gotten by using the `$()` operator we saw previously. It then takes that form and constructs a query string from all its elements. In this case the query string is added to the body of the request, as is typical when POSTing data elements. The last thing we do is specify the function to call when the request successfully completes, in this case an inline function. You can just as easily reference another stand-alone function, or function that is a member of some object—whatever you prefer. Prototype provides a number of callbacks like this, including `onComplete` and `onFailure`, so you can hook into the `XMLHttpRequest` lifecycle easily.

The `Ajax.Updater()` function is virtually identical; the only difference is that there is a parameter before the URL: the ID of an element on the page to update. The other difference is that typically you will not have `onSuccess` defined because `Ajax.Updater()` is essentially including its own version of that callback.

There is a fair bit more in Prototype than is used in this project, so I encourage you to explore it. One other frequent criticism of Prototype is that, unfortunately, documentation and support is sparse, and worse, the code is barely documented at all (although that is in line with the idea of keeping JavaScript as small as possible, so I for one am not sure it is such a bad thing). In fact, if you go to the main Prototype site (`http://prototype.conio.net`), I think you will be dismayed at how little is there. Fortunately, a gentleman named Sergio Pereira has put together a very good bit of documentation, which you can find here: `http://www.sergiopereira.com/articles/prototype.js.html#Enumerating`. It is well worth a few minutes to peruse this documentation. I think you will agree that, criticisms aside, Prototype has a fair bit to offer.

WebWork

In this book we have created a number of application frameworks to make the server component of the applications a little more flexible and consistent. We'll also be introduced to a "real" framework named Struts in the next chapter (although I would be willing to bet you already are familiar with Struts to some extent—most Java web developers are).

WebWork is a project under the OpenSymphony banner (`http://opensymphony.com`) that is somewhat similar to the Apache Foundation. WebWork, to quote directly from the WebWork site, is "a Java web-application development framework. It is built specifically with developer productivity and code simplicity in mind, providing robust support for building reusable UI templates, such as form controls, UI themes, internationalization, dynamic form parameter mapping to JavaBeans, robust client and server side validation, and much more."

WebWork is very similar to Struts in most respects. In fact, as of this writing, WebWork is undergoing incubation at Apache and will in fact become Struts 2.0 (which is now being called the Struts Action 2 framework). If you have any experience at all with Struts, even the little bit you would get from looking at AjaxChat in Chapter 9, then WebWork will be quite familiar to you. If you have never seen Struts before, don't worry; WebWork is really quite simple!

In WebWork, the basic unit of work is the `Action`. This is a class that will perform some function in response to some URL being called by a client. `Action`s in WebWork are also holders for incoming request parameters. As an example, look at this class:

```
package.apress.ajaxprojects.theorganizer.actions;
import webwork.action.*;
public class LogonAction {
  private String username;
  private String password;
  public void setUsername(String username) {
    this.username = username;
  }
  public void setPassword(String password) {
    this.password = password;
  }
  public String execute() {
    if (username.equals("bill") && password.equals("gates")) {
      return Action.SUCCESS;
   } else {
      return Action.ERROR;
    }
  }
}
```

This could be an `Action` in WebWork that responds to the user entering their credentials and clicking a Logon button on a page. So, how do we hook this up to a URL? By creating a file called `xwork.xml` and placing it in `WEB-INF/classes` of our webapp. The contents of this file might be something like this:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork 1.0//EN"
"http://www.opensymphony.com/xwork/xwork-1.0.dtd">
  <xwork>
```

```
  <include file="webwork-default.xml"/>
  <package name="default" extends="webwork-default">
    <default-interceptor-ref name="completeStack"/>
     <action name="logon"
       class="com.apress.ajaxprojects.theorganizer.actions.LogonAction">
       <result name="success">main.jsp</result>
       <result name="error">logon.jsp</result>
     </action>
  </package>
</xwork>
```

WebWork by default maps itself to all requests ending in `.action`. So, when the user clicks the Logon button, it submits a form to `logon.action`. WebWork intercepts this, and looks up logon in `xwork.xml`. It finds it, and instantiates the class named, `LogonAction` in this case. By default, WebWork will call a method named `execute()`, which is expected to return a `String`. The return value should map to one of the result names configured for this `Action`. In this case, if the user is valid (obviously this is not NSA-level security!), it returns the value of the `SUCCESS` constant found in the `Action` class in WebWork. This causes `main.jsp` to be returned. If the entered credentials are not valid, then `ERROR` is returned, which causes `logon.jsp` to be returned, which is presumably where the user started, instead.

One interesting feature of WebWork is autopopulation of incoming request parameters. That is how we were able to examine the username and password without doing anything. Before WebWork calls `execute()`, it takes all the request parameters, and calls setter methods on the `Action`, if they exist. In this case, it calls `setUsername()` and `setPassword()`, so that when `execute()` is finally called, the parameters are all there for us already; there's no need to directly access the request object as is typically necessary without a framework like this.

WebWork allows you to also specify what method to execute in the `Action` by adding a `method` attribute to the `<action>` element. In this way, you can group common functionality together in a single `Action` and call different methods depending on which `Action` mapping is called.

One other feature of WebWork that is used in The Organizer that I would like to make you aware of is the `ActionContext`. `ActionContext` is an object that is populated with various pieces of information on a per-request basis. For instance, you can get to the request object through the `ActionContext`. `ActionContext` is implemented as a `ThreadLocal`, which means there is a separate copy of its variables for each thread. The thing that makes this neat is that there is no need to pass `ActionContext` to any method in an `Action`, and indeed, WebWork will not. You can always access `ActionContext` by doing the following:

```
ActionContext context = ActionContext.getContext();
```

You could now get to request by doing the following:

```
HttpServletRequest request =
  (HttpServletRequest)context.get(ServletActionContext.HTTP_REQUEST);
```

The key point to this is that you can do this from *anywhere*. In other words, if your `Action` calls on another class, which itself calls another class, you can still use the previous code to get to things, as long as it all is executing within the same thread. This can actually be abused fairly easily because it means you might be tempted to go directly to the request object from a

business delegate class, for instance, which would not be good separation of concerns. As long as you can exercise discipline and not be tempted to do that, and in fact never use this capability outside of `Actions`, then what you get are very clean `Actions` that are Plain Old Java Objects (POJOs); there are no classes to extend, no interfaces to implement (an oft-stated failing of Struts is that you are forced to extend a base `Action` class, so you lose your one opportunity to extend). Note that the requirement to implement interfaces or extend classes isn't typically necessary in WebWork, although there *are* in fact interfaces to implement and classes to extend, all of which can provide additional capabilities and services automatically to your application, as we'll see in this code.

WebWork is a rather expansive framework covering far more than I can possibly describe here. WebWork in fact has some built-in Ajax functionality, including integration with DWR and Dojo out of the box. I encourage you to spend some time looking at the WebWork documentation to see what it offers. Especially if you have experience with Struts, since WebWork is the future of Struts, it would be a good idea to become familiar with it in my opinion. This project will give you a good basis to start with, because it does not delve too deeply into WebWork. In fact, aside from what we have seen here, the only other features used are some of the WebWork JSP tags, which we'll see later. As a gentle introduction to WebWork, though, The Organizer should serve you very well!

HSQLDB

HSQLDB, formerly called Hypersonic SQL, is a lightweight 100 percent pure Java SQL database engine. It supports a number of modes, including in-memory (for use with applets and such), embedded (which is how we'll be using it here), and client-server mode, which is basically a stand-alone database server. HSQLDB is the embedded database engine used in OpenOffice, and that is a pretty good pedigree if you ask me!

You can find HSQLDB's website here: `www.hsqldb.org`. One of the great things about HSQLDB is how drop-dead simple it is to use. First, one of its very convenient features is that if you attempt to access a database that does not exist, it will go ahead and create it for you. So, to get a database set up, this is all we have to do:

```
Class.forName(Globals.getDbDriver()).newInstance();
Connection conn = DriverManager.getConnection("jdbc:hsqldb:c:\temp\myDatabase",
  "sa", "");
conn.close();
```

This will create a new directory named `myDatabase` in `c:\temp`, and will create a basic database for us. At this point, we can go ahead and use standard JDBC and SQL to create tables, insert data, or whatever we want to do. There is no complex startup procedure and no setup code. There are not even any classes to import specific to HSQLDB!

HSQLDB is housed in a single JAR file with no outside dependencies. The JAR file is less than 1MB in size, so when they say "lightweight," they aren't kidding! Yet, it supports many features, such as views, temp tables and sequences, referential integrity, triggers, transaction support, Java stored procedures and functions, and even `Object` data types!

If you need to do database work in your application, and if you do not have or want a full-blown relational database management system (RDBMS), HSQLDB is a great solution. In fact, even if you *do* have a full-blown RDBMS like Oracle or SQL Server, you might want to think about HSQLDB anyway.

Spring JDBC

Unless you have been living under the proverbial rock for the past year or so, you have almost certainly heard of the Spring framework. You most likely heard about it first in the context of dependency injection, or IoC (Inversion of Control). This is probably what it is most well known for. However, Spring is much more than that.

Spring is what is termed a "full-stack" framework, meaning it pretty well covers all the bases a J2EE developer might need. It takes a layered approach, meaning that each "module" of functionality can be, more or less, used independently, and you can add only the functionality you need. Spring runs the gamut from dependency injection as mentioned, to web Model-View-Controller (MVC), to various general-purpose utility functions such as string manipulations and such, Object-Relational Mapping (ORM), and JDBC.

Speaking of JDBC, that is in fact the unit of functionality The Organizer directly uses (WebWork itself uses some other features, such as dependency injection, but the application code itself does not). The Spring JDBC package includes classes that make working with JDBC easier and, more important, less error-prone. One of the banes of JDBC programming is that developers are sometimes forgetful beasts, and neglect to do things like release database connections when they have finished with them. This leads to resource leaks, and eventually a crashed application, or worse yet, an entire server. With Spring JDBC, these types of mistakes are virtually impossible.

With Spring JDBC, database access basically boils down to two steps. First, get a data source to work with, and two, execute the pertinent SQL. For step one, the following code is used:

```
dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName(Globals.dbDriver);
dataSource.setUrl(Globals.dbURL);
dataSource.setUsername(Globals.dbUsername);
dataSource.setPassword(Globals.dbPassword);
```

The `Globals` class contains a number of constants used in The Organizer, including the details of connecting to the database, such as the driver to use, the URL, the username, and the password. Once we have a data source, step two is accomplished like this:

```
JdbcTemplate jt = new JdbcTemplate(dataSource);
List notes = jt.queryForList(
  "SELECT * FROM myTable"
);
```

The `JdbcTemplate` class is our gateway into the world of Spring JDBC. As you can see, it provides a `queryForList()` method, which returns us a `List` representing the result set. No more messing around with `ResultSet` objects! Other methods are provided, such as `queryForMap()`, which is used to retrieve a single record from the database and get a `Map` in return, making it very simple to access the fields of the record.

Note that I am not showing any connection-handling code. That is because the data source and template handle all that for us. No worries about closing the connection or dealing with statements and that sort of typical JDBC work.

One last thing I'd like to mention about Spring JDBC is that it wraps `SQLExceptions` in custom exception classes so that your code can be abstracted away from the standard JDBC classes entirely. As an example, when creating a new account in The Organizer, you could select a username that is already in use. In that case, the `JdbcTemplate` class will throw a `DataIntegrityViolationException` object, which we catch and handle appropriately.

The JDBC package is just one small part of Spring. A great deal more information is available to help you develop your applications better and faster. Take some time to check out what Spring has to offer; I think you will love what you find: `www.springframework.org`.

Visualizing the Finish Line

If you are familiar with the Macintosh family of computers from Apple, and more specifically, the more recent user interfaces they offer, then The Organizer will look familiar because the overall style is based on the "aqua" look and feel. Buttons and tabs have a bubble-shaped styling with a gentle gradient luster to them. Of course, in black and white in a book you cannot get the full effect, so go play a bit before continuing!

Now, let's familiarize ourselves with the application just a bit.

Figure 8-1 shows the Day At A Glance view. This is the first thing a user sees after they log on. It presents the user with the tasks and appointments for the current day.

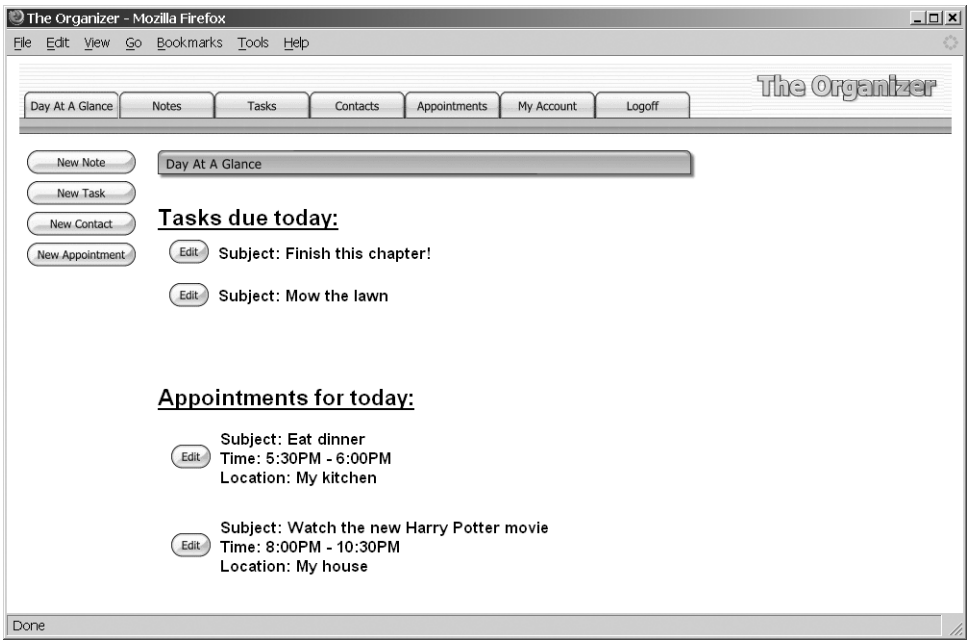

Figure 8-1. *The Organizer's Day At A Glance screen, the first thing seen by the user after logging on*

Along the top are tabs for the major areas of functionality: notes, tasks, appointments, contacts, and maintenance of your user account. You also have the ability to jump back to the Day At A Glance view at any time, as well as log off.

In Figure 8-2, we can see an example of creating a new task.

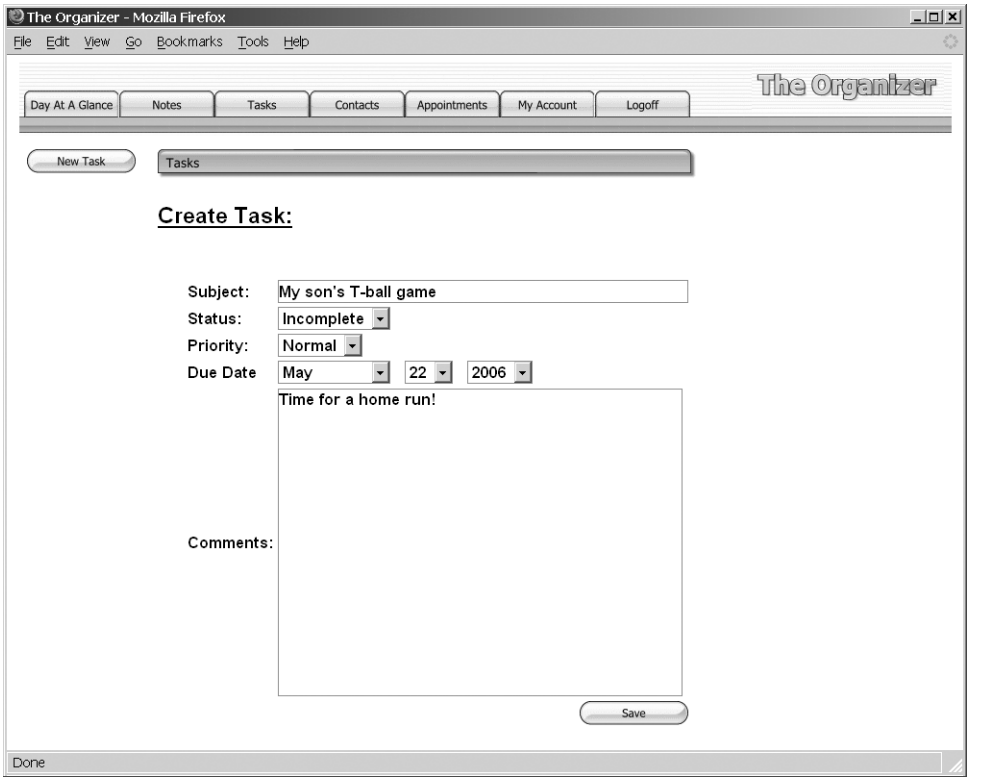

Figure 8-2. *Creating a new task*

Figure 8-3 shows editing an existing note. Creating a note looks virtually identical, except for different headings and so forth.

One last example is shown in Figure 8-4. This is the week view, showing all the appointments for the current week. All of the other views look the same; they just show different appointments, that is, appointments for the current month, day, or the entire year. You can select a date to use as the basis for this. So, if you wanted to see all the appointments for 2007, for instance, you can select any date in 2007, and the year view will show the appropriate appointments.

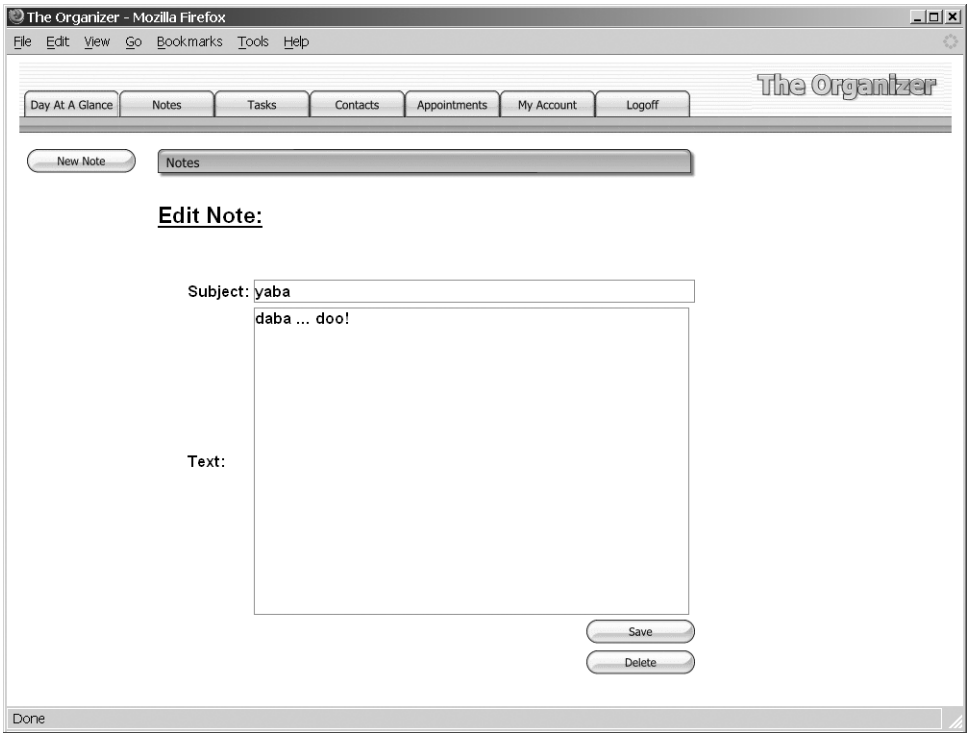

Figure 8-3. *Editing an existing note*

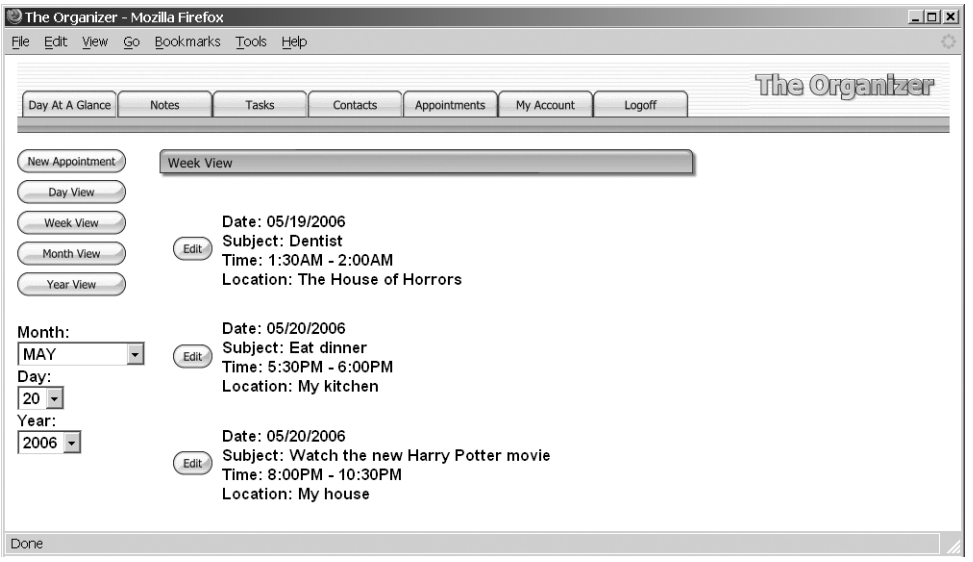

Figure 8-4. *Viewing appointments for the current week*

And now, without further ado: the code!

Dissecting the Solution

First, let's get a feel for the directory structure of The Organizer, shown in Figure 8-5.

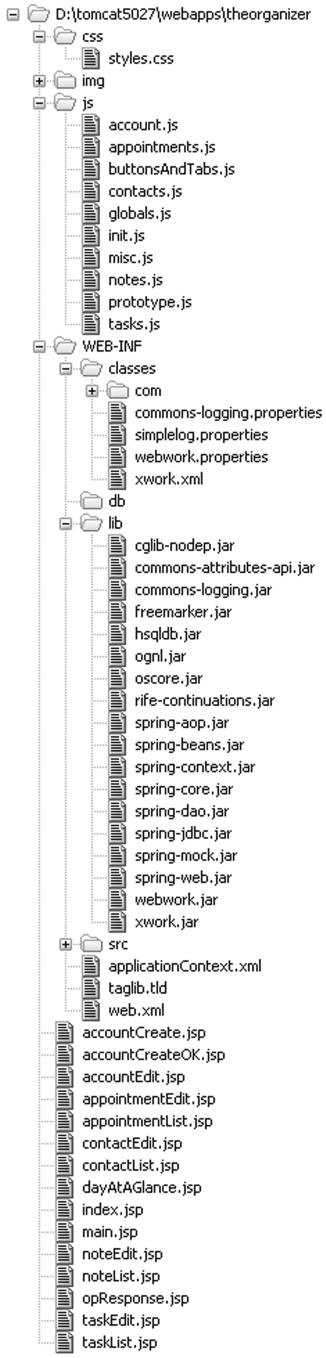

Figure 8-5. *Directory structure layout of The Organizer*

In the root directory we have a number of JSPs, which we'll be looking at shortly. In the `/css` directory is our stylesheet, `styles.css`. The `/img` directory, which is not expanded here to conserve space, contains all the image resources, including image rollovers, headers, and so forth. The `/js` directory contains all the JavaScript used in this application. In `/WEB-INF` we find the usual `web.xml`, plus `applicationContext.xml`, which is the Spring framework configuration file. We also find `taglib.tld`, the taglib descriptor for the WebWork tags. One new thing here that we haven't seen in any other projects in this book is the content of `/WEB-INF/classes`, most notably, `xwork.xml` and `webwork.properties`. Both of these are configuration files for WebWork. They need to be accessible via a class loader and so cannot be in `/WEB-INF` and must instead be in `/WEB-INF/classes` (or somewhere else accessible via a class loader, although this is the typical location). We also see two files, `commons-logging.properties` and `simplelog.properties`, which combine to configure logging for the application. Our source files, as usual, are found in `WEB-INF/src`, which is not shown here but is pretty much the same as every other source directory in this book.

Finally, the `/WEB-INF/lib` folder contains all the libraries that The Organizer depends on; they are listed in Table 8-1.

Table 8-1. *The JARs That the Organizer Depends On, Found in WEB-INF.lib*

JAR	Description
`cglib-nodep.jar`	Required by WebWork. CGLib is a powerful, high-performance, and high-quality code generation library. It is used to extend Java classes and implements interfaces at runtime.
`commons-attributes-api.jar`	Required by WebWork. Commons Attributes provides a runtime API to metadata attributes such as doclet tags, inspired by the Nanning and XRAI projects as well as JSR 175 and C# attributes.
`commons-logging.jar`	Required by WebWork. Jakarta Commons Logging is an abstraction layer that sits on top of a true logging implementation (like Log4J), which allows you to switch the underlying logging implementation without affecting your application code. It also provides a simple logger that outputs to `System.out`, which is what this application uses.
`freemarker.jar`	Required by WebWork. FreeMarker is a "template engine," a generic tool to generate text output (anything from HTML to autogenerated source code) based on templates. It's a Java package—a class library for Java programmers. It's not an application for end users in itself, but something that programmers can embed into their products.
`hsqldb.jar`	HSQLDB, our embedded SQL database engine.
`ognl.jar`	Required by WebWork. OGNL stands for Object-Graph Navigation Language. It is an expression language for getting and setting properties of Java objects. You use the same expression for both getting and setting the value of a property.
`oscore.jar`	Required by WebWork. OSCore is a set of utility classes that are common to the other components of OpenSymphony. Contains essential functionality for any J2EE application.

continued

Table 8-1. *Continued*

JAR	Description
rife-continuations.jar	Required by WebWork. RIFE/continuations is a subproject of RIFE that aims to make its support for continuations in pure Java available as a general-purpose library.
spring-aop.jar	Spring framework AOP (Aspect-Oriented Programming) package. Core Spring AOP interfaces, built on AOP Alliance AOP interoperability interfaces.
spring-beans.jar	Spring framework Spring beans package. This package contains interfaces and classes for manipulating JavaBeans.
spring-context.jar	Spring framework context package. This package builds on the beans package to add support for message sources and for the Observer design pattern, and the ability for application objects to obtain resources using a consistent API.
spring-core.jar	Spring framework core classes package. Provides basic classes for exception handling and version detection, and other core helpers that are not specific to any part of the framework.
spring-dao.jar	Spring DAO (Date Access Objects) package. Exception hierarchy enabling sophisticated error handling independent of the data access approach in use.
spring-jdbc.jar	Spring JDBC package. The classes in this package make JDBC easier to use and reduce the likelihood of common errors.
spring-mock.jar	Spring mock objects package.
spring-web.jar	Spring web package. Includes various web-related classes.
webwork.jar	The WebWork JAR, where most of WebWork lives.
xwork.jar	XWork is a command pattern framework that is used to power WebWork as well as other applications. XWork provides an Inversion of Control container, a powerful expression language, data type conversion, validation, and pluggable configuration.

The Client-Side Code

Let's begin by looking at the configuration files, beginning with `web.xml`. The four elements that are of importance are these:

```
<!-- WebWork filter. -->
<filter>
  <filter-name>webwork</filter-name>
  <filter-class>
    com.opensymphony.webwork.dispatcher.FilterDispatcher
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>webwork</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

```
<!-- Spring IoC. -->
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<!-- The Organizer initialization. -->
<listener>
  <listener-class>
    com.apress.ajaxprojects.theorganizer.listener.ContextListener
  </listener-class>
</listener>

<!-- WebWork taglib. -->
<taglib>
  <taglib-uri>webwork</taglib-uri>
  <taglib-location>/WEB-INF/taglib.tld</taglib-location>
</taglib>
```

The first filter mapping is what makes WebWork "go." Instead of using a servlet, as Struts and most other MVC frameworks do, WebWork uses a filter. One thing of note is that even though the filter is mapped to `/*`, which means all incoming requests are handled by the filter, in fact by default only requests ending with `.action` will be handled by WebWork. The "real" filtering is done within the filter itself.

The listener is for Spring functionality needed by WebWork to function. Likewise, the second listener is needed by The Organizer. Some onetime initialization is performed by this listener, as we'll see when we get to that class.

Lastly, we need a declaration of the WebWork taglib, which references the TLD in `WEB-INF`. This is nothing but your standard, everyday taglib definition file, so I will not be reviewing it here.

applicationContext.xml

The `applicationContext.xml` file that is also found in `WEB-INF` is the Spring configuration file. Because The Organizer does not use Spring's dependency injection capabilities, this file is essentially empty, but it is still required to be present.

webwork.properties

The other configuration files are located in `WEB-INF/classes`. First up is `webwork.properties`. This is a standard Java properties file where you can change various default settings of WebWork. The only one you will find in this particular file is `webwork.objectFactory = spring`, which tells WebWork to use Spring for creating objects.

commons-logging.properties, simplelog.properties

Two other files of interest are `commons-logging.properties` and `simplelog.properties`. Jakarta Commons Logging (JCL) is used by The Organizer for its logging functions, and further, the `SimpleLog` is used, which is a logger that JCL provides that simply writes its output to `STDOUT`.

These two files allow us to configure that. First, in `commons-logging.properties`, we find the single line

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
```

This is what informs JCL to use `SimpleLog`. In `simplelog.properties`, we have some settings for this logger:

```
org.apache.commons.logging.simplelog.defaultlog=info
org.apache.commons.logging.simplelog.log.com.apress.ajaxprojects=info
```

This says that, by default, only messages of level `info` and higher will be logged by this logger. We then further say that any class that is in the package `com.apress.ajaxprojects`, or any child package of that package, will be logged if the message is of level `info` or higher. It is important to set both a default level and specific level for the classes in The Organizer because other classes in the application may also use JCL, and if they do, we want to be able to control their logging level separately. For instance, WebWork itself uses JCL, and if the default level is changed to `debug`, for instance, you will see a great deal more messages logged coming from WebWork. If you wish to see more messages from The Organizer, set the second line to a level of `debug` or `trace`. It is helpful to be able to set this level during development as much more will be caught and displayed.

xwork.xml

The last file you will find in this directory is perhaps the most important: `xwork.xml`. If you have any experience with Struts, this is analogous to `struts-config.xml`. If you have never used Struts, then simply put, `xwork.xml` is the file that maps incoming request URIs to specific `Action` classes. That is at least its most important purpose.

`xwork.xml` is a fairly lengthy file, so I will only show a small portion of it here:

```
<!-- Retrieve a list of all notes (initial notes view). -->
<action name="noteList"
  class="com.apress.ajaxprojects.theorganizer.actions.NoteAction"
  method="list">
  <result name="success">noteList.jsp</result>
</action>

<!-- Show the create note view. -->
<action name="noteCreateShow"
  class="com.apress.ajaxprojects.theorganizer.actions.ForwardAction">
  <result name="success">noteEdit.jsp</result>
</action>

<!-- Create (create) a new note. -->
<action name="noteCreate"
  class="com.apress.ajaxprojects.theorganizer.actions.NoteAction"
  method="create">
  <result name="success">opResponse.jsp</result>
  <result name="error">noteEdit.jsp</result>
</action>
```

These three blocks are referred to as `Action` mappings. The first mapping is used when we want to see a list of notes for the current user. This mapping is saying that when a request comes in for the resource `noteList.action`, the class `NoteAction` should be instantiated, and the `list()` method should be called. This method will be expected to return a `String` that matches one of the `<result>` elements. `success` is one of the typical return strings. Notice there are two different results for the `noteCreate` mapping. This is so that if any sort of error occurs when saving the note (which is when this mapping would be executed), then the `noteEdit.jsp` page would be returned, presumably with some sort of error message for display to the user.

With the configuration files out of the way, let's now turn our attention to the source files, starting with the client side of the house.

styles.css

This is of course the stylesheet for The Organizer, and there is not really much to it. We have a grand total of nine selectors. The first is `cssTopNav`, which styles the upper portion of the screen where the tabs appear. Note the use of a background image that is repeated horizontally across the `<div>` this will ultimately be applied to. `cssSideNav` is the area where the buttons appear on the left-hand side. `cssMain` is for the main content in the middle. Note the usage of the four padding attributes. This sets a few pixels of space around all four edges so that the buttons and the tab bar at the top do not bump into the main content. Next we find the `cssDAAGItem` selector, which styles the tasks and appointments seen on the Day At A Glance screen. Likewise, `cssDAAGHeading` is the tasks and appointments header that defines each section. `cssScrollContent` is also used to style the main content in order to give us a scrolling `<div>`. This is done with the goal of avoiding the entire page scrolling if the content is too large vertically (it still can depending on your screen resolution, but that is largely unavoidable without resorting to frames). `cssError` is used to show the errors that can occur when logging on. Finally, `cssInput0` and `cssInput1` are used to style all input elements in the application. The one ending with 0 is what the element will look like what it does not have focus, and the one ending with 1 is what it looks like when it has focus (a different background color).

index.jsp

This is the default welcome page as defined in `web.xml` and serves as our logon page. We start with a normal taglib declaration for the WebWork tags. After that is our stylesheet import, and an import of the JavaScript file `buttonsAndTabs.js`. We'll look at this in more detail later, but for now suffice it to say that the code required to do the rollovers of the buttons is in it. After that we have a `<script>` block containing two variables: `tabsButtonsEnabled` and `rolloverImages`. The former determines whether or not the buttons are enabled, and the latter will store the preloaded images for the buttons. Those images are loaded on page load by calls to the `createRolloverImages()` function. In short, this function accepts the name of a button and loads the images for it and stores them in the `rolloverImages` array.

After that we see the following block:

```
<ww:if test="message!=null">
  <div class="cssError"><ww:property value="message" /></div>
</ww:if>
<ww:else>
  <div class="cssError"> </div>
</ww:else>
```

This is our first encounter with the WebWork tags. The `<ww:if>` tag is used to perform a logic comparison. Recall that when a URI mapped to WebWork comes in, an `Action` is executed. This `Action` can contain fields that correspond to incoming request parameters, as well as any other fields we wish, which can be used in the JSP. For instance, in the `AccountAction`, which is the `Action` that will be called to create a new account, we find a field named `message`. This is used to communicate errors to the user when logging on—if the username is not found, for instance. This `<ww:if>` tag is simply checking whether or not that field is `null`. WebWork knows to go look for the field in the current instance of the `AccountAction` class; we do not have to be anymore specific than naming the field. If the field it not `null`, we use the `<ww:property>` tag to display the value. If it is `null`, we output a nonbreaking space HTML entity so that the space reserved for the message is still present (to avoid the input fields being pushed down when a message is displayed).

After this block we come across a `<ww:form>` tag. This is the WebWork version of the standard `<form>` tag. We specify the URL to submit the form to, as well as the style class to apply to the form. Within the form we find a `<ww:textfield>` tag, which renders an `<input type="text">` tag.

Note the value attribute of these tags. For example, on the tag for the username, we find `%{username}`. This is how you specify fields in the `Action` to populate the value attribute with.

We lastly have a `<ww:submit>` tag; this one is of type `image`.

If you are observant, you may be scratching your head here a bit… how exactly do all the fields line up so nicely on the screen when there is no hint of structure in the markup? The answer is that the WebWork tags render a table structure for the form for us! Note the `label` attribute on the fields—this is the first column of a table. The field itself is the second column. WebWork does all this for us by default. WebWork supports various "themes," which change what gets rendered by the tags. The default theme is `XHTML`, which is what is rendered here (take a look at the generated source!). The other theme you will see in this project is `simple`. This basically tells WebWork to not render the table and such for us, but to just render the form field tag. There is also an `Ajax` theme, which provides automatic Ajax functionality on the form.

accountCreate.jsp

This is the JSP that is shown when the user clicks the Create Account button on the logon screen. It is quite similar to `index.jsp`. In fact, if you compare the two, they are virtually identical, so I will forgo looking over it here.

accountCreateOK.jsp

This is the JSP that renders the server's response when an account has been successfully created. It is nothing more than a message saying the account has been created, and a button that, when clicked, leads the user to the Day At A Glance view. The same JavaScript used for the buttons on the `accountCreate.jsp` is used here for the button and rollover on it.

main.jsp

This JSP is loaded after a user has been validated during logon, or when they click the OK button after creating an account. It is the unchanging content of The Organizer, including the tabs at the top and the buttons along the side. It is responsible for loading all the JavaScript required by the application in one go.

In fact, after the importing of the stylesheet, we find a block of 10 JavaScript file imports, nine for The Organizer itself, and one for the Prototype library.

After that, the markup begins. The markup can be logically divided into three sections: the top navigation bar, the side navigation bar, and the main content area.

For each of the tabs along the top navigation bar, we find a very similar chunk of markup:

```
<td valign="bottom"><img src="img/notes0.gif" id="notes"
  onClick="showNotes();"
  onMouseOver="rollover(this);"
  onMouseOut="rollout(this);"></td>
```

We have ourselves a table cell with an image inside it. The image of the tab starts off in its nonhighlighted (0) state. We give it an ID to refer to later, and attach the appropriate `onClick` event handler. We also assign `onMouseOver` and `onMouseOut` handlers using the `rollover()` and `rollout()` JavaScript functions that we'll see shortly.

After that we find a section that looks almost identical to the tabs, but this time it is for the buttons along the side. The interesting thing to note is that all the buttons are always present from the start. The appropriate buttons for a given view are shown, while the rest are hidden. Also in this section you will find some `<select>` elements. These are used to select a date when looking at appointments. Note that unlike the buttons and tabs, they have `onFocus` and `onBlur` handlers as well. This allows us to have the background change colors when a given control gets focus. The `onChange` event is also hooked so that any change in these fields results in the current view being updated (if applicable).

Lastly, we have a table cell with two `<span>` tags within it, one with the ID `mainContent` and one with the ID `pleaseWait`. The `mainContent` `<span>` is where the result returned by the server for any of our Ajax calls will be placed. Only this section will be dynamically updated; everything else remains the same (I am not counting the buttons being shown and hidden according to the view—I am referring to the fact that the rendered content remains the same everywhere but in `mainContent`). The `pleaseWait` `<span>` is where the Please Wait message is. These two `<span>`s are alternately displayed and hidden as required. Note that `<span>` is used instead of `<div>` to avoid unwanted line breaks after each, important for making the layout work as expected.

dayAtAGlance.jsp

The Day At A Glance screen is the first thing a user sees when they log on to the application, and this is the JSP responsible for rendering it. More specifically, when `main.jsp` loads, it calls a JavaScript function `init()`, which makes an Ajax request for this view. This JSP renders the response, and it is displayed in the `mainContent` `<div>`. The Day At A Glance view shows tasks that are due today, as well as appointments for today.

Let's examine the code that renders the tasks due today:

```
<div class="cssDAAGHeading">Tasks due today:</div>
<ww:if test="%{!tasks.isEmpty()}">
  <ww:iterator value="tasks">
    <form>
      <input type="hidden" name="createdDT" value="<ww:property
        value="createdDT"/>">
```

```
        <table border="0" cellpadding="0" cellspacing="0" class="cssDAAGItem">
          <tr>
            <td width="1">
              <input type="image" src="img/edit0.gif" id="edit"
                align="absmiddle" onmouseover="rollover(this);"
                onmouseout="rollout(this);"
                onclick="taskRetrieve(this.form);return false;">
            </td>
            <td width="10"> </td>
            <td>
              Subject: <ww:property value="subject" />
            </td>
          </tr>
        </table>
      </form>
    </ww:iterator>
  </ww:if>
  <ww:else>
    There are no tasks to display
  </ww:else>
```

Here we meet a new WebWork tag: `<ww:iterator>`. This tag iterates over a collection that is a field in the current `Action`, in this case, the collection of tasks for today. Before that iteration begins, though, we check to see if the collection is empty. If it is, the `<ww:else>` tag executes, and we render a message saying there are no tasks to display. If it is not empty, however, we iterate over the collection. For each element, we output some markup. Notice the use of the `<ww:property>` tag again, and also notice how we only have to name the field to reference. This name is always a field of the current object from the collection. There is no need to give the current object a name, as is typical with other taglibs. The WebWork tags are smart enough to know that the name specified by the `value` attribute of the `<ww:property>` tag is a member of the current object from the collection. Cool!

The code that renders the markup for the appointments is basically the same, except that of course there are different fields to display, such as start time, end time, and location.

accountEdit.jsp

This is a simple JSP that is rendered as the result of an Ajax request when the user clicks the My Account tab. It is just a simple form that gives the user the ability to change their password, since this is currently all that can be edited (the username is essentially the key of the table, so it cannot be changed).

opResponse.jsp

This JSP renders the response to many different Ajax events, including saving notes, setting (or deleting) tasks and appointments, or creating new items. It handles all of these by going through a batch of `if` blocks to tailor itself appropriately to the operation that was performed. For instance, one of the things that is different depending on what operation was performed is the graphical header that is displayed. WebWork is kind enough to put the requested URI in the

request as an attribute, so we can interrogate that value to see what operation was performed. So, if we are doing something with an appointment, for instance, this block of code will execute:

```
if (requestURI.indexOf("appointment") != -1) {
  headerFile = "appointments"; whatItem = "Appointment";
  targetFunc = "showAppointments();";
}
```

The `headerFile` variable is the name of the header graphic, minus path and extension. `whatItem` is text that will be displayed, such as "Appointment has been created". `targetFunc` is the name of the JavaScript function that will be executed when the user clicks the OK button.

The other variable that has to be set is `whatOp`, which is what operation was just performed. The code that does this is as follows:

```
if (requestURI.indexOf("create") != -1) { whatOp = "created"; }
if (requestURI.indexOf("update") != -1) { whatOp = "updated"; }
if (requestURI.indexOf("delete") != -1) { whatOp = "deleted"; }
```

By inserting these four variables' values into the markup, we can generate an output appropriate for the operation that was just performed, and make the OK button go to the appropriate view afterward. It's very handy to have this all in one file rather than a separate JSP for each response… imagine, three JSPs for each unit of functionality: one JSP for updating, one for deleting and one for creating, and those three types for each of the note, task, contact, and appointment function groups. Twelve JSPs versus a single one with some branching logic—I know I prefer the one!

appointmentEdit.jsp, contactEdit.jsp, noteEdit.jsp, taskEdit.jsp

I am grouping these four together because, essentially, they are all the same. True, they have different input fields on them, but the basic structure and function is identical in all of them. These are the JSPs used to show the forms for editing or creating an appointment, contact, note, or task.

We start out with a block of code to put the appropriate heading on the page:

```
<div class="cssDAAGHeading">
<%
  String requestURI = (String)request.getAttribute("webwork.request_uri");
  requestURI = requestURI.toLowerCase();
  if (requestURI.indexOf("create") != -1) {
    out.println("Create Note:");
  } else {
    out.println("Edit Note:");
  }
%>
</div>
<br>
```

We again do this by interrogating the request URI that led to this JSP, and render the appropriate text, "Create Note" or "Edit Note".

After that is a typical WebWork-based input form. Since this page will function as both edit and create, we need to output the values found in the `Action` instance as the value of the various form fields so that when we are editing we start out with the current values. In the `Action` itself we find that we have a `NoteObject` that stores all the values for the note. This has a subject field. To set the value of the text field to that value, we use the `%{note.subject}` notation. Note the reference to the `note` field, and then the `subject` field in that object.

As I mentioned, all four of these JSPs are essentially the same; the only difference relates to the fields present in the rendered form. Have a look at all of them to see this for yourself.

I will, however, point out one section in the `taskEdit.jsp` that is of interest:

```
<tr>
  <td><label>Due Date</label></td>
  <td>
    <ww:select name="dueMonth" theme="simple"
      list="#{'':'', '01':'January', '02':'February', '03':'March',
      '04':'April', '05':'May', '06':'June', '07':'July', '08':'August',
      '09':'September', '10':'October', '11':'November',
      '12':'December'}"
      cssClass="cssInput0"
      onfocus="this.className='cssInput1';"
      onblur="this.className='cssInput0';" />
     
    <ww:select name="dueDay" theme="simple"
      list="#{'':'', '01':'01', '02':'02', '03':'03', '04':'04', '05':'05',
      '06':'06', '07':'07', '08':'08', '09':'09', '10':'10', '11':'11',
      '12':'12', '13':'13', '14':'14', '15':'15', '16':'16', '17':'17',
      '18':'18', '19':'19', '20':'20', '21':'21', '22':'22', '23':'23',
      '24':'24', '25':'25', '26':'26', '27':'27', '28':'28', '29':'29',
      '30':'30', '31':'31'}"
      cssClass="cssInput0"
      onfocus="this.className='cssInput1';"
      onblur="this.className='cssInput0';" />
     
    <ww:select name="dueYear" theme="simple"
      list="#{'':'', '2006':'2006', '2007':'2007', '2008':'2008',
      '2009':'2009', '2010':'2010', '2011':'2011', '2012':'2012',
      '2013':'2013', '2014':'2014', '2015':'2015', '2016':'2016',
      '2017':'2017', '2018':'2018', '2019':'2019', '2020':'2020'}"
      cssClass="cssInput0"
      onfocus="this.className='cssInput1';"
      onblur="this.className='cssInput0';" />
  </td>
</tr>
```

Recall earlier when discussing WebWork in general when I mentioned how the WebWork tags were nice enough to render a table structure for us so that our forms automatically take on a nice, clean appearance. This was a result of the default XHTML theme. However, in the case of the due date for a task, I wanted to have three `<select>` dropdowns in a row. This is not

possible by default because after the first one, the WebWork tag would close the table row. Instead, I had to set the theme of the tags to `simple`, which makes WebWork just render the input element, not the table code. I take responsibility for that. In this way, I can have my three `<select>` dropdowns in a row, and it is still nice and clean code.

Also notice the use of the `<ww:select>` tag. This tag allows you to populate the options for the dropdown from a collection in the `Action`, or with a literal list, as is done here. I think this also makes the code cleaner, because the alternative is a batch of `<option>` tags, as in a typical `<select>`, which are usually put on a separate line each. So with the `<ww:select>` tag, we wind up with fewer lines of code, and fewer characters typed, which is always good for us lazy programmers! Being able to reference a collection in the `Action` is even cooler, but to me it only makes sense if you have a dynamic list, or something read from a database, for instance. For static, known lists such as these, I find it just as easy to put them in the JSP itself. Your mileage may vary!

This JSP actually provides the capability to create a new note or edit an existing note, but it also provides the necessary markup to delete a note. This is done by a simple check at the end to see whether or not the requested URI was created. If it wasn't, it means we are editing an existing item. In that case, the Delete button is rendered.

appointmentList.jsp, contactList.jsp, noteList.jsp, taskList.jsp

As with the edit pages, the list JSPs are all pretty much the same. If you've seen one, you've seen 'em all, as the saying goes! So, let's get the "if you've seen one" part out of the way:

```
<%@ taglib prefix="ww" uri="webwork" %>

<img src="img/head_notes.gif">
<br><br>

<div class="cssScrollContent">

  <ww:if test="%{!notes.isEmpty()}">
    <ww:iterator value="notes">
      <form>
        <input type="hidden" name="createdDT" value="<ww:property
          value="createdDT"/>">
        <input type="image" src="img/edit0.gif" id="edit" align="absmiddle"
          onmouseover="rollover(this);" onmouseout="rollout(this);"
          onclick="noteRetrieve(this.form);return false;">
         
        Subject: <ww:property value="subject" />
      </form>
    </ww:iterator>
  </ww:if>
  <ww:else>
    There are no notes to display
  </ww:else>

</div>
```

That's it, completely. It is nothing more than iteration over a `List` in the `Action`, and generating markup for each item (or generating text saying there are no notes to display). Everything here we have seen before; it should just about be second nature by this point!

One thing to note is the `return false;` at the end of the `onClick` event handler. You will see this throughout the code for this project. The reason for it is that Internet Explorer will do a double-submit when you click these buttons: the one manually done by virtue of the call to `noteRetrieve()`, or whatever the event handler calls, and one for the automatic form submission. Returning false indicates to the browser that the automatic submission should not occur. Interestingly, this is not necessary in Firefox.

globals.js

The `globals.js` file contains a small handful of global variables, four in fact. The first, `rolloverImages`, stores the preloaded images for all the button and tab rollovers used in the application. This is an associative array, so we can pull out the images for a given button, for instance, by referencing it by name. No need to worry about index numbers! The next variable, `tabsButtonsEnabled`, tells us whether the user can click buttons or tabs at the moment. This is used to disable buttons and tabs when an Ajax request is working. Next is the `currentView` variable, which as its name implies, tells us what view we are currently looking at: `taskView`, `noteView`, `accountView`, and so on. The variable `subView` is used for appointments to record whether we are looking at day view, week view, month view, or year view when viewing appointments.

init.js

This source file contains a single function, `init()`, amazingly enough! This function is called from the `onLoad` event handler of the `main.jsp` page. Its job is, unsurprisingly, to initialize the application on the client. It performs three basic functions.

First, it preloads all the images for buttons and tabs. To do this, it makes use of a function `createRolloverImages()`, found in `buttonsAndTabs.js`. It accepts the name of the button or tab to load. It then creates two `Image` objects for it and loads the appropriate GIF file into it. It then adds these images to the `rolloverImages` array, keyed by name. This saves us from a lot of repetitive image preload code.

Second, it sets the `<select>` dropdowns in the left-hand side of the screen to the current date. These dropdowns are used when dealing with appointments to set the current date, which is then used as the basis for all the appointment views. This code makes use of the function `locateDDValue()` found in `misc.js`, which locates and selects a specified value in a specified `<select>` element. Note that this code uses the Prototype `$()` function to reference the dropdowns, like so:

```
locateDDValue($("dsMonth"), month);
```

Third, `init()` throws up an alert dialog to welcome the user to the system, and then calls `showDayAtAGlance()`, which fires off an Ajax request to get the result of the Day At A Glance service.

misc.js

This file contains a handful of, well, miscellaneous functions, which did not fit elsewhere. First up is `showPleaseWait()`, which is called whenever an Ajax request is made. It inserts the Please Wait message into the `mainContent` area.

Second is `showDayAtAGlance()`. This fires off an Ajax request to get the contents of the Day At A Glance view. Let's look at this function:

```
function showDayAtAGlance() {

  if (tabsButtonsEnabled) {
    showPleaseWait();
    new Ajax.Updater("mainContent", "dayAtAGlance.action", {
      method : "post",
      onSuccess : function(resp) {
        currentView = "dayAtAGlance";
        setupSidebarButtons("newNote", "newTask", "newContact",
          "newAppointment");
        hidePleaseWait();
      },
      onFailure : ajaxError
    });
  }

} // End showDayAtAGlance().
```

You will find that all of the Ajax calls in The Organizer have the same basic form. First, we only fire the request if `tabsButtonsEnabled` is true. When it is false, it means an Ajax request is already processing. It is set to false by the call to `showPleaseWait()`, which hides the main content and shows the Please Wait message (the `pleaseWait` `<span>` we saw earlier in `main.jsp`). Next, we use the `Ajax.Updater()` function call. `Updater()` is a convenience method that performs what is probably the most common Ajax function: updating some element on the page, usually a `<div>`, with the response from the server. The first argument passed to this function is the ID of the element to update. The second argument is the URL to send the request to. After that comes a collection of arguments, including what method to use, `POST` in all cases in this application; the function to call when the response is successfully returned (usually an anonymous function listed inline, as is shown here); and a function to execute when a failure occurs, in this case a call to `ajaxError()`, which is also found in `misc.js` (it is just an alert message saying an error has occurred). Note the call to `hidePleaseWait()` in the `success` handler. This shows whatever the server returned, which is now in the `mainContent` `<span>` in `main.jsp`, and also reenables the buttons and tabs.

After that we find `logoff()`, which is called when the user clicks the logoff tab. This does nothing but set the location of the window to `logoff.action`, which calls an `Action` that terminates the session and forwards to `index.jsp`, where the user can log on again if they wish.

Lastly, `locateDDValue()` is used to find and locate a value in a given `<select>` element. This is just a simple iteration over the options in the `<select>`, and when the matching element is found (ignoring case), it is selected.

buttonsAndTabs.js

This file contains all the code that makes the tabs and buttons work. There are four functions in it. First up is `setupSidebarButtons()`. This is called any time the view changes, that is, when a response to an Ajax request comes back from the server. Its job is to enable the appropriate buttons. Passed into this function is a variable argument list. What this means is that you could call this function any of the following ways:

```
setupSidebarButtons();
setupSidebarButtons("newNote");
setupSidebarButtons("newNote", "newTask");
setupSidebarButtons("newNote", "newTask", "newContact");
setupSidebarButtons("newNote", "newTask", "newContact", "newAppointment");
```

Implicitly available to every JavaScript function is an object named `arguments`. You access it by doing `xxxx.arguments`, where `xxxx` is the name of the function. So, in this case we do `var args = setupSidebarButtons.arguments;`. This object is an array of the arguments passed to the function. In the case of the first call earlier, this would be an empty array; in the second call, an array with a single element; and so on. So, in this function, we iterate over the array, and for each, enable the named button, like so:

```
  for (i = 0; i < args.length; i++) {
    $(args[i]).style.display = "block";
  }
```

In this way, the code that calls this function can set up whichever buttons are appropriate without having to explicitly pass an array or have a set number of arguments that this function accepts, each corresponding to a specific button. Variable argument lists are a powerful feature of JavaScript, and one that is used frequently in various libraries you may encounter.

The next function we find is `createRolloverImages()`, which we have seen used previously. Now we can see what it actually does:

```
function createRolloverImages(inName) {

  var img = new Image();
  img.src = "img/" + inName + "0.gif";
  rolloverImages[inName + "0"] = img;
  img = new Image();
  img.src = "img/" + inName + "1.gif";
  rolloverImages[inName + "1"] = img;

} // End createRollover().
```

Recall that this function is passed the name of a button or tab to load images for. So, using that name, we instantiate two `Image` objects, and then proceed to load them with the images whose name we form by taking the name passed in and appending either 0 or 1 to the end, 0 for the normal nonhover version and 1 for the hover version. We then add these two `Image` objects to the `rolloverImages` array that we saw in `Globals.js`. Recall that this is an associative array, and we'll reference the images by name, including the 0 or 1 at the end.

Lastly in this source file are the two functions that actually make use of the `rolloverImages` array, `rollover()` and `rollout()`. These correspond to the `onMouseOver` and `onMouseOut` events

of all buttons and tabs. They both work the same way: first, we check to see if buttons and tabs are enabled, and only continue if they are. That way, we can lock out the buttons and tabs from being clickable by the user when an Ajax request is processing. Next, we see if we were passed a DOM ID of the button or tab as the second argument to this function (the first being a reference to the button or tab object itself). This ID is actually the name of the image file, sans the 0 or 1 at the end, which is appended by the two functions to form the full filename. The reason the ID is passed manually here is because of a bug in WebWork at the time of this writing where the ID would not be rendered on these elements in the forms. So, any of the buttons that are part of a form—that is, those in the main content area—would not work with this function because all this function should need is the object reference, from which the ID can be retrieved. Since it is not being rendered, however, this will not work. So, in some cases the ID is manually passed in, and in other cases it is not (it is passed in manually on WebWork-generated buttons). If no ID was passed in, we grab it from the object.

Finally, using this ID, we get the appropriate image from the `rolloverImages` array, and update the button (or tab) with it.

account.js

This JavaScript file contains the code that deals with accounts. Three functions are present: `showMyAccount()`,`accountUpdate()`, and `accountDelete()`. All of them have the same basic form as the `showDayAtAGlance()` function that we saw previously. `showMyAccount()` is the function called when the user clicks the My Account tab, and it returns the markup for the form to edit the account. `accountUpdate()` is called to save the updated account (which is just the ability to change the password in actuality). Lastly, `accountDelete()` is called when the user clicks the Delete Account button.

appointments.js, contacts.js, notes.js, tasks.js

I have lumped all four of these together again because they are all basically the same, so just reviewing one is probably sufficient.

In each of these files you will find seven functions. Using `notes.js` as the example, we first find `showNotes()`, which is called when the user clicks the Notes tab. This makes an Ajax call via Prototype in the same manner as we saw in `showDayAtAGlance()` and uses the `Ajax.Updater()` function to display the returned result, which is a list of notes in this case. After that is `doNewNote()`, which is called when the user clicks the New Note button. This displays the form generated by `noteEdit.jsp`. Next is `noteCreate()`, which makes the call to save a new note as a result of the user clicking the Save button when they have previously clicked New Note. After that comes `noteRetrieve()`, which is called when the user clicks the Edit button next to a note they wish to edit. It again displays the result of the rendering of `noteEdit.jsp`. After that is `noteUpdate()`, which updates an existing note when the user clicks Save on the note editing screen. After that is `noteDelete()`, called when the user wants to delete a note. The result of both of the previous functions is that `opResponse.jsp` is rendered and returned, tailored to the function that was performed as previously described. Finally is the function `validateNote()`, which performs some validations before submitting a note, either a new one or one being edited.

As mentioned, the other three files are quite the same, but please do examine them anyway to see that for yourself. There are a few exceptions to this. First, in `appointments.js`, you will find that in the `appointmentRetrieve()` function, I use `Ajax.Request()` rather than `Ajax.Updater()`. This is because simply updating the `mainContent` `<div>` is not sufficient

because there are <select> elements that must have a value selected. By using AjaxRequest() instead, we have complete control over what happens when the response is successful, and in this case that means using the previously described locateDDValue() function to select a value in those <select> fields. Also in appointments.js are a few additional functions, namely doWeekView(),doMonthView(), and doYearView(). These are the functions that are called when the applicable button is clicked to change the current view. This makes the same call as does showAppointments(), which could be thought of as showDayView() because that is what it is doing. We pass to the server the name of the view we want, and it obliges with the applicable data. Also, you will find a dsSelectorChange(), which is the function called by the onChange event of the date selector on the side. This simply calls showAppointments(), doDayView(), doMonthView(), or doYearView(), depending on what the current view is, so that the data can be updated using the selected date as the basis.

Also note that the functions listed, aside from the initial showNotes() function and the validateNote() function in our example, make up a typical CRUD (Create, Retrieve, Update, Delete) collection of functions that are the hallmark of many data entry applications, which is essentially what this is after all. This CRUD pattern is repeated on the server side, as we'll now see.

The Server-Side Code

With the client-side code out of the way, we can now begin to explore what makes The Organizer tick on the server side of the fence.

Globals.java

The Globals class is a simple holder of static values such as database driver name, database username and password, and so on. The only thing of real interest here is the dbURL field, which is the only nonfinal field. This is of course the URL string to use for the database. The issue is that this value has to be constructed dynamically because it is a file system path to the WEB-INF/db directory. This path is constructed in ContextListener.java and then set in Globals so it is available to the rest of the application.

ContextListener.java

This class is executed when the context starts up to do some basic application initialization. There are basically three important tasks it performs. First, it constructs that dbURL value mentioned in describing Globals.java using the following code:

```
String dbURL = "jdbc:hsqldb:" +
  inEvent.getServletContext().getRealPath("/WEB-INF") +
  "/db/theorganizer";
Globals.setDbURL(dbURL);
```

As mentioned before, this constructs a real file system path to the WEB-INF/db directory, and it also names the database, theorganizer specifically.

Next it creates the database, if it does not already exist. The following code accomplishes that:

```
Class.forName(Globals.getDbDriver()).newInstance();
Connection conn = DriverManager.getConnection(Globals.getDbURL(),
  Globals.getDbUsername(), Globals.getDbPassword());
conn.close();
```

HSQLDB does its thing here and creates the database if it does not already exist; otherwise it just initializes the database engine.

The last step is to create the database tables. This is accomplished with a series of calls to the `createTable()` method of the `TableDAO` class, which we'll see later. We pass to this method the name of the table to create, using the constant table names declared in `Globals.java`. As a prelude to looking at that DAO, the `createTable()` method will retrieve metadata about the database, and check to see if the named table exists. If it does, it returns right away, and if it does not exist, the method goes ahead and creates the table. All of the database setup is encapsulated and automatic in the code, no there are no SQL scripts or things like that to run.

AccountObject.java

The `AccountObject` class is the class that represents a user account. It is a simple JavaBean with two fields: username and password. As with all the other object classes, as well as all the `Action` classes, my typical `toString()` method is present so that we can easily display the contents of an instance of these classes. Otherwise, this is a rather boring class!

AppointmentObject.java, ContactObject.java, NoteObject.java, TaskObject.java

I once again have chosen to lump all these together because they are quite similar. I could have easily listed `AccountObject` here as well since they are all nothing but plain old JavaBeans. As such, they are not worth discussing very much except to say that they each map to a record in the corresponding database table: `appointments`, `contacts`, `notes`, or `tasks`. Speaking of database tables…

TableDAO.java

The `TableDAO` object is a data access object that has one purpose in life: to create the database tables The Organizer needs to function. It is used exactly once, at application startup, and is called from `ContextListener`.

In the constructor, a data source is created, as we saw when we discussed HSQLDB. This is typical of all the DAOs in this application. After that, we have a single method, `createTable()`, which accepts as an argument the name of the table to be created. It begins its work by getting metadata on the database, like so:

```
Connection       conn  = dataSource.getConnection();
DatabaseMetaData dbmd  = conn.getMetaData();
ResultSet        rs    = dbmd.getTables(null, null, "%", null);
boolean          found = false;
while (rs.next()) {
  String s = rs.getString(3);
  if (s.equalsIgnoreCase(inTableName)) {
    found = true;
  }
}
rs.close();
```

If the table requested to be created is found, we'll not do anything after this. If it is not found, though, we do some branching logic based on which table was requested, and the

appropriate SQL is executed to create the table. For instance, to create the accounts table, we use the following code:

```
log.info("Creating " + inTableName + " table...");
JdbcTemplate jt = new JdbcTemplate(dataSource);
if (inTableName.equalsIgnoreCase(Globals.TABLE_ACCOUNTS)) {
  jt.execute(
    "CREATE TABLE accounts ( " +
    "username VARCHAR(20), " +
    "password VARCHAR(20) " +
    ");");
  jt.execute("CREATE UNIQUE INDEX username_index ON accounts (username)");
```

Here we see a standard `CREATE TABLE` query, and then a bit of SQL to create a unique index on the username field. This disallows creating an account that already exists.

There are a total of five tables: `accounts`, `notes`, `tasks`, `contacts`, and `appointments`. Figure 8-6 shows a basic schema diagram of the database.

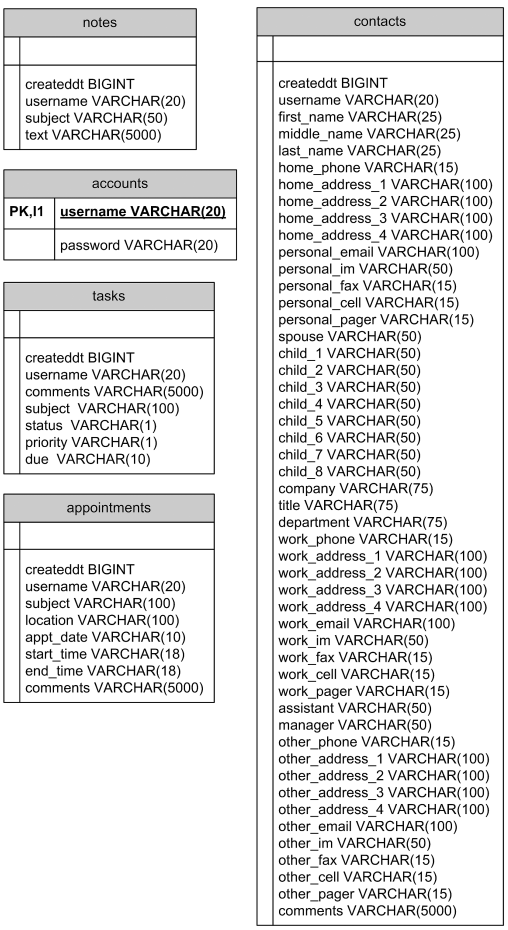

Figure 8-6. *Simple database schema diagram of The Organizer's database*

AccountDAO.java

The AccountDAO class deals with operations for user accounts. Specifically, we again see the CRUD pattern emerging: we have an accountCreate() method, an accountRetrieve() method, an accountUpdate() method, and an accountDelete() method. All but the accountRetrieve() method accepts a single argument, an AccountObject instance. The accountRetrieve() method accepts a username to look up. All but the accountRetrieve() method are simple SQL calls using the Spring JdbcTemplate class we have previously seen.

accountRetrieve() is not much more, but let's have a quick look at it:

```
public AccountObject accountRetrieve(final String inUsername) {

  log.debug("AccountDAO.accountRetrieve()...");

  log.debug("username to retrieve : " + inUsername);
  JdbcTemplate jt = new JdbcTemplate(dataSource);
  List rows = jt.queryForList(
    "SELECT * FROM accounts WHERE username='" + inUsername + "'"
  );
  AccountObject account = null;
  if (rows != null && !rows.isEmpty()) {
    account = new AccountObject();
    Map m = (Map)rows.get(0);
    account.setUsername((String)m.get("USERNAME"));
    account.setPassword((String)m.get("PASSWORD"));
  }
  log.info("Retrieved AccountObject : " + account);

  log.debug("AccountDAO.accountRetrieve() Done");
  return account;

} // End accountRetrieve().
```

This too uses JdbcTemplate to query for the record matching the requested username. We use the queryForList() method and then grab the first element of the List, which should contain only a single record. We create an AccountObject from it, populating the username and password from the Map in the List. queryForList() returns a List of Maps, where each Map represents a row in the result set, but since it is a Map, it is a bit easier to deal with. The created object is then returned, or null is returned if the matching record was not found.

AppointmentDAO.java, ContactDAO.java, NoteDAO.java, TaskDAO.java

Yet again, I have chosen to group four classes together because they are virtually identical in the end. These are the four DAO classes that deal with appointments, contacts, notes, and tasks, respectively. As in the AccountDAO class, you will find four methods in each class corresponding to the CRUD operations.

You will also find one additional method in each, an xxxList() method, where xxx is the object we are dealing with: appointment, contact, note, or task. Each of these methods performs a query to get a list of items for the user, and returns a list of the appropriate object type,

either AppointmentObject, ContactObject, NoteObject, or TaskObject. Each of them accepts a username argument.

In addition, in the TaskDAO class, the listTasks() method also accepts a boolean that tells it to either return only those tasks due today or return all of them. When true is passed, only tasks due today are returned, and this is used when rendering the Day At A Glance view. When false is passed, all tasks are returned, which is used when rendering the view the user sees when they click the Tasks tab.

Likewise, the listAppointments() method is a bit different. In addition to username, it accepts a Date object, and a string telling it which view to return (day, week, month, or year). The Date object is used as the basis for the view. This method performs a number of steps to generate the appropriate list, so let's walk through those steps.

First, a list of all appointments for the user is obtained from the database:

```
log.debug("Servicing '" + inViewType + "' view...");
log.debug("username to retrieve appointments for : " + inUsername);
JdbcTemplate jt = new JdbcTemplate(dataSource);
List appointments = jt.queryForList(
  "SELECT createddt, subject, location, start_time, end_time, appt_date " +
  "FROM appointments WHERE username='" + inUsername + "' order by " +
  "appt_date, start_time"
);
```

Next, the day, month, year, and week of month are extracted from the date passed in:

```
GregorianCalendar gc = new GregorianCalendar();
gc.setTimeInMillis(inDate.getTime());
int day         = gc.get(Calendar.DATE);
int month       = gc.get(Calendar.MONTH) + 1;
int year        = gc.get(Calendar.YEAR);
int weekOfMonth = gc.get(Calendar.WEEK_OF_MONTH);
```

Note that the month value is incremented by 1. This is because the month returned from the gc.get(Calendar.MONTH) call is 0–11, but the month stored in the database is 1–12, so to be able to do comparisons later, we need the value to be in the range 1–12.

The code then begins to iterate over the result set List. For each, it extracts the appointment's date and extracts from that the same information as was extracted from the date passed in, as shown in the previous code. This is a bit different because the date is actually stored as a character string in the database, so we have to do some string parsing to get the same information:

```
String appointmentDate = (String)m.get("APPT_DATE");
int m_month = Integer.parseInt(appointmentDate.substring(0, 2));
int m_day = Integer.parseInt(appointmentDate.substring(3, 5));
int m_year = Integer.parseInt(appointmentDate.substring(6, 10));
int m_weekOfMonth = (new GregorianCalendar(
  m_year, m_month - 1, m_day)).get(Calendar.WEEK_OF_MONTH);
```

The variable m here is the Map of the current row in the result set. Now that we have the same information for both the date passed in and the date of the appointment we are currently examining, we can do some simple comparisons to determine if the appointment

should be shown in the requested view, and if it should, we set the variable takeIt to true. For instance, if the day view was requested, the following logic is performed:

```
if (inViewType.equalsIgnoreCase("day")) {
  if (day == m_day && month == m_month && year == m_year) {
    takeIt = true;
  }
}
```

There is one such block of logic for each view type—a total of four if blocks. Then comes one last block, whose job it is to create a new AppointmentObject and populate it if the appointment is to be shown:

```
if (takeIt) {
  AppointmentObject appointment = new AppointmentObject();
  appointment.setCreatedDT(((Long)m.get("CREATEDDT")).longValue());
  appointment.setUsername(inUsername);
  appointment.setAppointmentDate((String)m.get("APPT_DATE"));
  appointment.setSubject((String)m.get("SUBJECT"));
  appointment.setLocation((String)m.get("LOCATION"));
  appointment.setStartTime((String)m.get("START_TIME"));
  appointment.setEndTime((String)m.get("END_TIME"));
  appointmentsOut.add(appointment);
}
```

The code is added to the appointmentsOut List, and is returned, and that is how the list of appointments for a given view is generated.

ForwardAction.java

In some cases in the application—for instance, when the user clicks the New Account button—we just want to return the outcome of a JSP rendering without actually doing any other work, essentially forwarding the request directly to a JSP. In this case, we still want the request to go through the WebWork framework as this is generally considered a best practice. The ForwardAction is a simple Action that is used in these situations. All it does is returns Action.SUCCESS, which causes the <forward> named success to be forwarded to. There is no actual work to be done, so this Action is in effect just a dummy Action so that we can still use the framework to make the request. Struts has a similar construct, also named ForwardAction.

LogonAction.java

The LogonAction class, not surprisingly, handles when the user is attempting to log on. It uses the default execute() method, which means that WebWork will execute this method without being told to. The real work of this Action is encapsulated in the following code snippet:

```
// Get the user.
AccountDAO    dao     = new AccountDAO();
AccountObject account = dao.accountRetrieve(username);
```

```
// See if they exist, and if their password is correct.
if (account == null) {
  log.info("User not found");
  message = "User not found";
  return Action.ERROR;
} else {
  if (!password.equalsIgnoreCase(account.getPassword())) {
    log.info("Password incorrect");
    message = "Password incorrect";
    return Action.ERROR;
  }
}
```

It calls on the `AccountDAO` to get the account for the `username` that came from the request parameters (remember, WebWork will have called `setUsername()` on this class before calling `execute()`, so the `username` field now has the parameter value). If it is not found, the appropriate message is set in the `Action`, and `Action.ERROR` is returned, which returns us to `index.jsp`, where the message field is used to display the error. If the account is found, and the password is incorrect, we return the appropriate message there (I realize this is not a security best practice, but we are not trying to build Fort Knox here!). If the account is found and the password is correct, `Action.SUCCESS` will be returned, and the `AccountObject` is also put in session, where it will be used later to get the username for SQL queries (so the username is never passed again as a request parameter).

LogoffAction.java

I will give you just one guess what this `Action` is for. Yes, it handles when the user wants to log off! The pertinent code is rather short and sweet here:

```
ActionContext      context = ActionContext.getContext();
HttpServletRequest request =
  (HttpServletRequest)context.get(ServletActionContext.HTTP_REQUEST);
HttpSession        session = request.getSession();
session.invalidate();
```

Recall in our earlier discussion of WebWork we talked about the `ActionContext`—how it is a `ThreadLocal` and how it is easy to get a reference to it, and through it, to the usual suspects: `HttpServletRequest`, `HttpSession`, and so on. That is precisely what we do here. The only job to be performed here is to invalidate the session, at which point the user is directed to `index.jsp` and is "logged off."

AccountAction.java

The `AccountAction` handles all operations required to work with user accounts. Just like in the DAOs, there are methods corresponding to the typical CRUD operations, and in the `Actions` (not just this one, but the `AppointmentAction`, `ContactAction`, `NoteAction`, and `TaskAction` classes as well) they are literally named `create()`, `retrieve()`, `update()`, and `delete()`.

Each one looks suspiciously like the code you saw in the LogonAction. This is no accident. The fundamental structure is essentially the same for all these Actions and all these methods. One difference you will spot is in the create() method:

```
if (!password.equalsIgnoreCase(password_2)) {
  log.debug("Password not matched");
  message = "Password not matched";
  return Action.ERROR;
}
```

The typical "enter your password twice to verify" paradigm is used, and this is where that validation occurs.

Another difference is when we ask the AccountDAO to actually add the new account to the database. In this case, the possibility exists that the username is already in use, in which case an exception will be thrown. Remember that Spring will wrap the underlying SQLException into something more generic, so we have to catch that and handle it accordingly:

```
AccountDAO dao = new AccountDAO();
try {
  dao.accountCreate(account);
} catch (DataIntegrityViolationException dive) {
  // Username already exists.
  log.debug("Username already exists");
  message = "That username already exists.  Please try another.";
  return Action.ERROR;
}
```

DayAtAGlanceAction.java

This Action is called to show the contents of the Day At A Glance view. It boils down to simply calling two different DAOs to get the list of tasks due today as well as the list of appointments for today:

```
// First, get the list of tasks due today.
TaskDAO taskDAO = new TaskDAO();
tasks = taskDAO.taskList(username, true);

// Next, get the list of appointments for today.
AppointmentDAO appointmentDAO = new AppointmentDAO();
appointments = appointmentDAO.appointmentList(username, new Date(), "day");
```

Nothing to it!

AppointmentAction.java, ContactAction.java, NoteAction.java, TaskAction.java

Last but not least are the four Actions that deal with appointments, contacts, notes, and tasks. I have decided to group them together because of their extreme similarity to one another, as well as to the other Actions we have already looked at. As with the other Actions, you will again find the four CRUD methods, and they are essentially identical; just the SQL queries and

objects they work with differ. Note that there are no validations performed in any of these. There is very little input validation to be performed at all outside of account creation, and those validations are handled in the client code (not the most robust design, I acknowledge, but sufficient for our purposes).

You will also find in each a `list()` method, which returns an appropriate list of the object corresponding to the item the Action is from. You will also find one more method in each, named `getXXXXObject()`, where `XXXX` is `Appointment`, `Contact`, `Note`, or `Task`. The purpose of this method is to save some redundant code. When creating or updating an item, we need to construct the appropriate object: `AppointmentObject`, `ContactObject`, `NoteObject`, or `TaskObject`. When we are creating a new appointment, we want to populate the object with the input from the user. When updating an existing appointment, the DAO will do this for us from the database. However, in both cases, some bits of information will be the same, such as username. So, to avoid having this code redundantly in both methods, I elected to break it out to a separate method. When editing an item, there is no harm in the object being populated from the input parameters; the fields in the `Action` will in fact be `null`s and initial values, but they will be overwritten with what the DAO gets from the database, so there is no harm in it.

Just to be sure there are no surprises, let's look at an example of each of these methods from the `NoteAction` class. First, `create()`:

```
public String create() {

  log.info("\n\n-------------------------------------------------------------");

  log.debug("NoteAction.create()...");

  // Display incoming request parameters.
  log.info("NoteAction : " + this.toString());

  // Call on the NoteDAO to save the NoteObject instance we are about to
  // create and populate.
  NoteObject nte = getNoteObject();
  NoteDAO    dao = new NoteDAO();
  // Need to override the createdDT that was populated by getNoteObject().
  nte.setCreatedDT(new Date().getTime());
  dao.noteCreate(nte);

  log.debug("NoteAction.create() Done");

  return Action.SUCCESS;

} // End create().
```

Nope, nothing surprising there. We get a `NoteObject`, and then a `NoteDAO` instance. We set the `createdDT` field of the `NoteObject` to the current time, and call `noteCreate()` in the DAO, passing it the `NoteObject`. Simple!

How about the `retrieve()` method?

```
public String retrieve() {
```

```
    log.info("\n\n-------------------------------------------------------");

    log.debug("NoteAction.retrieve()...");

    // Display incoming request parameters.
    log.info("NoteAction : " + this.toString());

    // Retrieve the note for the specified user created on the specified date
    // at the specified time.
    AccountObject account  = (AccountObject)session.get("account");
    String        username = account.getUsername();
    NoteDAO       dao      = new NoteDAO();
    note = dao.noteRetrieve(username, createdDT);
    log.debug("NoteAction : " + this.toString());

    log.debug("NoteAction.retrieve() Done");

    return Action.SUCCESS;

  } // End retrieve().
```

Here we have just a little bit more going on. We get the username from the `AccountObject` in session, and we pass that along to the `noteRetrieve()` of the `NoteDAO` instance we create. `note`, which is a field of the `Action` of type `NoteObject`, is set to the object returned by the call. Again, very simple.

Any chance that `update()` has much more going on?

```
  public String update() {

    log.info("\n\n-------------------------------------------------------");

    log.debug("NoteAction.update()...");

    // Display incoming request parameters.
    log.info("NoteAction : " + this.toString());

    // Call on the NoteDAO to save the NoteObject instance we are about to
    // create and populate.
    NoteObject nte = getNoteObject();
    NoteDAO    dao  = new NoteDAO();
    dao.noteUpdate(nte);

    log.debug("NoteAction.update() Done");

    return Action.SUCCESS;

  } // End update().
```

Nope, wouldn't appear so! This is even a bit simpler: a call to `getNoteObject()` gets us a `NoteObject` with all the input parameters already populated, and then it is nothing more than passing that object along to the `noteUpdate()` method of the DAO. Piece of cake!

Next we look at the `delete()` method:

```
public String delete() {

  log.info("\n\n-------------------------------------------------------------");

  log.debug("NoteAction.delete()...");

  // Display incoming request parameters.
  log.info("NoteAction : " + this.toString());

  // Call on the NoteDAO to delete the NoteObject instance we are about to
  // create and populate.
  AccountObject account  = (AccountObject)session.get("account");
  String        username = account.getUsername();
  NoteDAO       dao      = new NoteDAO();
  NoteObject    nte      = new NoteObject();
  nte.setCreatedDT(createdDT);
  nte.setUsername(username);
  dao.noteDelete(nte);

  log.debug("NoteAction.delete() Done");

  return Action.SUCCESS;

} // End delete().
```

Once more we find that it is sweet and to the point. To delete any given item we have to supply the DAO with the username and the `createdDT` value, which as you recall serves as essentially the unique key for any item. More specifically, the username and `createdDT` serve as something of a composite key. To remind you, the `createdDT` is the date and time the item was created in milliseconds. So, for a given user, it is virtually impossible that two items of the same type could ever get the same `createdDT` value.

The `list()` method is also very simple:

```
public String list() {

  log.info("\n\n-------------------------------------------------------------");

  log.debug("NoteAction.list()...");

  AccountObject account  = (AccountObject)session.get("account");
  String        username = account.getUsername();
  NoteDAO       dao      = new NoteDAO();
  notes = dao.noteList(username);
  log.debug("NoteAction : " + this.toString());
```

```
    log.debug("NoteAction.list() Done");

    return Action.SUCCESS;

  } // End list().
```

Just get the username from the `AccountObject` in session and pass it along to the `noteList()` method in the DAO, and we are done!

Just for the sake of completeness at this point, let's take a look at the `getNoteObject()` method:

```
  private NoteObject getNoteObject() {

    AccountObject account  = (AccountObject)session.get("account");
    String        username = account.getUsername();
    NoteObject    nte      = new NoteObject();
    nte.setCreatedDT(createdDT);
    nte.setUsername(username);
    nte.setSubject(subject);
    nte.setText(text);
    return nte;

  } // End getNoteObject();
```

Hopefully that is pretty much exactly what you expected to see!

Again, all of these methods are very similar in all four of these `Actions`. The differences lie in what fields are present in the `Action`, and by extension, what fields these methods deal with, and of course what objects they work with, and what DAOs—all the things you would expect to be different. The basic, underlying structure and flow of the code is the same, though, with one exception: the `list()` method of the `AppointmentAction`.

```
  public String list() throws ParseException {

    log.info("\n\n-------------------------------------------------------------");

    log.debug("AppointmentAction.list()...");

    AccountObject    account  = (AccountObject)session.get("account");
    String           username = account.getUsername();
    AppointmentDAO   dao      = new AppointmentDAO();
    Date             d        = null;
    SimpleDateFormat sdf      = new SimpleDateFormat();
    sdf.applyPattern("MM/dd/yyyy");
    d = sdf.parse(month + "/" + day + "/" + year);
    appointments = dao.appointmentList(username, d, view);
    log.debug("AppointmentAction : " + this.toString());
```

```
    log.debug("AppointmentAction.list() Done");

    return Action.SUCCESS;

  } // End list().
```

Here, some additional code is required to take the month, day, and year selected in the date selector on the left side of the screen and make a `Date` object out of it. Not exactly a big deal, but I wanted to point it out nonetheless.

Well, that about does it! We have now examined all the code that makes up The Organizer. I don't know about you, but I need a vacation!

Suggested Exercises

The Organizer is not meant to compete with the likes of Lotus Notes or Microsoft Outlook. Indeed, that level of functionality is far beyond what could be described in a book such as this. However, with just a few enhancements, The Organizer could offer a lot more to its users than the basics it offers now. Here are some suggestions for you to try that will challenge your newly gained knowledge of Prototype, HSQLDB, WebWork, and Spring:

- Integrate The Organizer and InstaMail. It would be nice if you could have one unified address book (the one in The Organizer is more robust and is the natural choice), and it would also be nice if you could send an e-mail at the click of a button while looking at the contact list in The Organizer.
- Graphical views for the appointments would allow you to see your appointments for a given time span.
- A busy search function would let you choose any user(s) in the database, and see whether they have appointments scheduled for a given time period.
- Allow for automated e-mail notifications when tasks are due and when appointments will soon begin. The code from InstaMail should be easy transportable to accomplish this. For bonus points, send e-mails to your cell phone's e-mail address so you can get your alerts on the go. Most cell providers offer this functionality nowadays; check with them for the details.

Summary

In this chapter we constructed something that is genuinely useful to anyone with a hectic lifestyle. If you were to host this application on an Internet-facing server, the ability to have access to your contacts, to-do lists, notes, and appointments from virtually anywhere would be a very handy thing. This chapter also discussed the Prototype library, which underpins a great many Ajax libraries. We also explored the WebWork application framework, which is set to become the next Struts. And last but not least, we explained how to use an embedded database named HSQLDB in our applications.